BITS PILANI K.K. BIRLA GOA CAMPUS

A Report on

Assignment 1: Genetic Algorithm for 3-CNF Problem

Prepared by

Shubhankar Kate **2018B4A70786G**

in fulfillment of the requirements of
CS F407 (Artificial Intelligence)

INTRODUCTION

This report outlines the various approaches that were tried and tested as part of Assignment 1 of the Artificial Intelligence (CS F407) course.

Some basic assumptions are taken:

- Instead of True, False values, each assignment that is generated consists of 1's and 0's (True and False respectively)
- The final model is printed as per the expectation of the assignment i.e if the assignment is 1 then literal is positive, 0 then literal is negative.

Problem Statement :

The problem statement defines a 3-CNF propositional formula consisting of 50 literals such that negation is depicted by negative numbers and the rest as positive numbers. Hence a possible formula can look like the figure given below.

$$(a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee \neg e \vee \neg d) \wedge (\neg b \vee c \vee d) \wedge (\neg c \vee \neg d \vee e)$$

For a clause of the form ($\sim a_{32}$ V $\sim a_{12}$ V a_{48}), it will be denoted as (-32 V -12 V 48).

Here we have to devise an optimized Genetic algorithm such that for a random sentence having exactly 50 literals, we are able to find an assignment such that it satisfies maximum clauses in a randomly generated sentence (within a 45 second time limit).

An Overview of Genetic Algorithm:

A Genetic Algorithm is an exploration/exploitation-based method based on the principles of Heredity and Natural Selection. It is often used to discover ideal or close ideal answers for difficult problem statements which otherwise would take up a lot of time and resources. We start with a population of potential answers for the given problem statement. These solutions then undergo recombination and mutation, creating new offspring, and this happens over what we call generations. Every individual is allotted a fitness value and the fitter offspring are allowed a higher opportunity to reproduce and yield even better offspring. This is repeated until a stopping point is reached (which varies from problem to problem). Genetic Algorithms are sufficiently randomized in nature, and they perform much better than trying various random solutions, as they exploit historical information.

APPROACHES TRIED (Q3)

Various components of the Genetic Algorithm were tried and tested in various ways. Some parts are given below

- Fitness function
- Selection of Parents
- Mutation
- Reproduction
- Generating new Generation

Fitness function:

The fitness function for this problem statement was defined in the assignment itself. In my research, I did not found any alternate/better implementation for it. Hence that remains pretty straightforward. For each assignment in a population, we check in each clause, if at least one literal is satisfied. In my code, for each sentence, the number of clauses satisfied is returned, and when calculating for a population, we take the percentage satisfied as to the return value.

Selection of Parents:

Originally I tried to randomly select parents, but that was giving only decent results. Then I thought of giving some kind of preference based on how good the parent is. Hence, parents are selected based on how well they fit the sentence generated. This is done by using an integral power of the fitness function and averaging over the entire population such that it gives values between 0 and 1. This is then used in the "*random.choice"* function for choosing in a random yet greedy way. After trying various powers for the fitness function, I settled on using power 50 of the fitness value. This is just to push the better fitting parents further in the list of being selected and hence optimizing the basic random choice approach.

Reproduction:

We choose a random crossover point for crossing over data over between two parents. We tried to optimize this by checking if the crossover generates a better child among the two that are generated, but this did not affect the accuracy much and took up a little extra time. Hence we stuck with crossing over at a random point. The function returns both children and it is not selected at this stage. All the children are considered in the next generation.

Something which I also tried was making an offspring array consisting of both parents and both children, sorting them according to their fitness value, and then returning the best 2 among them. This took a lot of time and gave a decrease in the accuracy. Hence choosing better candidates is only done when parents and children are combined when we create a new generation.

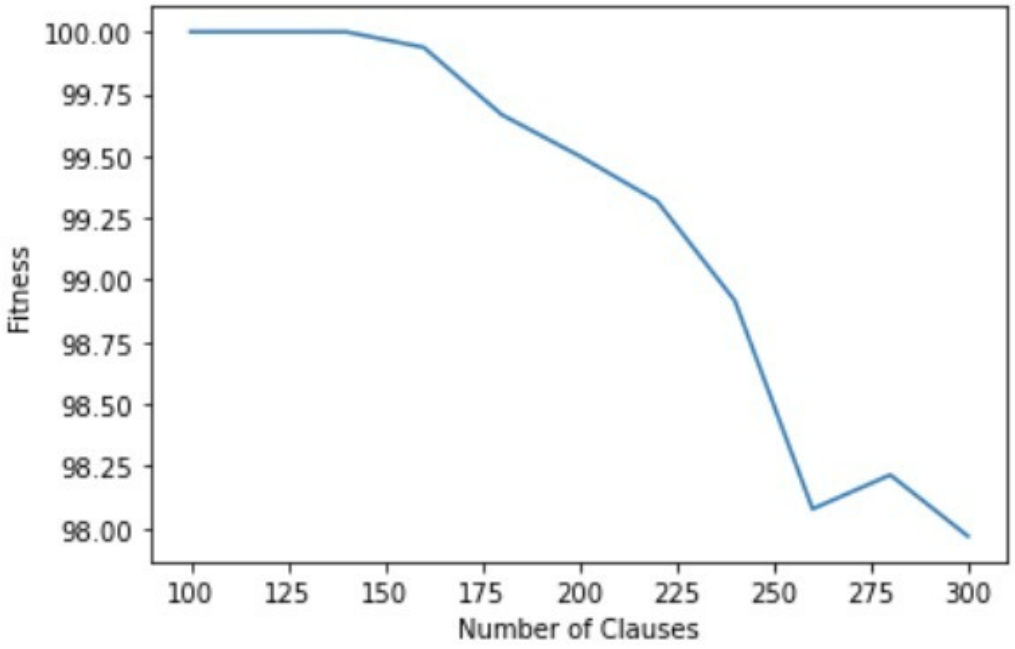
Mutation:

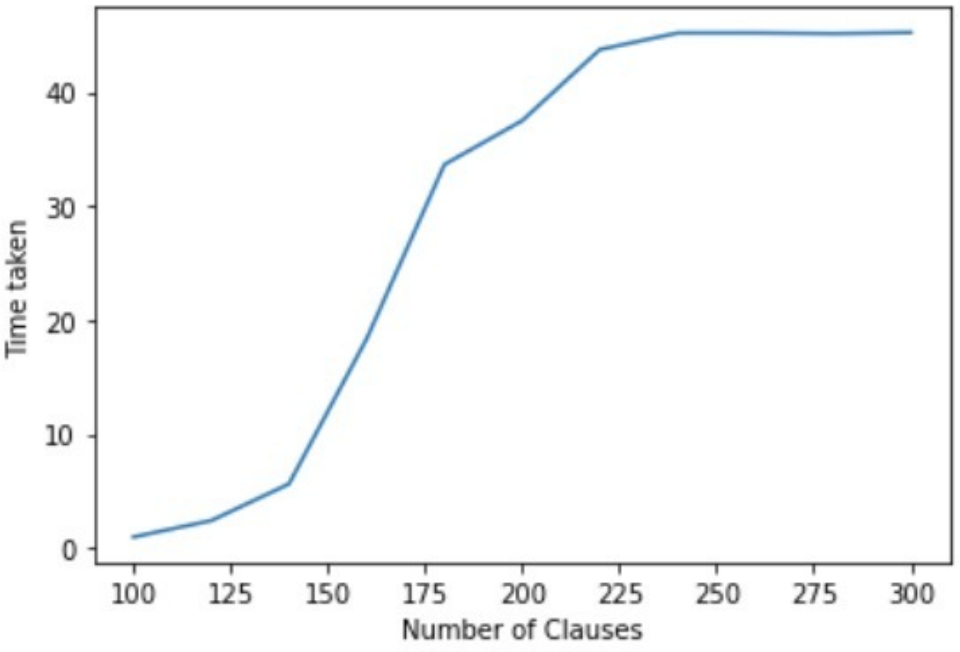
The earlier approach was to generate a random number k, which denotes the number of bits that are toggled during mutation. Then we can generate k unique random bits and toggle them. This approach took a lot of time and further decreased the accuracy of the algorithm. Then, we tried to mutate at every stage, but this failed to give good results for more than 5 bits. Hence, I stuck with just toggling a single bit at every stage which gave a considerable boost in both time and accuracy. Just to add a factor of randomness to this, we set the probability of mutation to 95%, which maintained the accuracy of the model. The percentage was tried by a random trial and error method. Percentage below 20% did not yield better results, and we noticed that on average we have to keep a higher percentage(close to 90%) as mutation at every stage is almost always beneficial. This is because the algorithm does not explore as much (for this problem statement) and tends to converge very fast. This can lead to getting stuck in local maxima, and hence a high mutation rate can curb this issue. It also encourages more exploration (as this algorithm converges by itself), helping in exploring more parts of the population to reach the most optimal solution.

Generating a new generation:

For each generation, we choose parents according to the aforementioned points. We first tried to use the generation made of children produced from the reproduction entirely as the next generation, but that only gave decent results. Hence we thought of using the parents as well as the children. Hence if the parents are better candidates for appearing in the next generation, they also have a chance to be chosen. This gave a very big boost in the accuracy. Then to again add a factor of randomness, we select k of them randomly using the preference order we used for parents (k is the size of the original population). A basic stopping criterion is added which stops the generations when accuracy reaches 100%.

GRAPHS OF FITNESS AND TIME TAKEN VS NUMBER OF CLAUSES (Q1 & Q2)





For each value of m, we ran 10 iterations and took an average over those 10 fitness and time values to obtain the above graphs.

DIFFICULTIES FACED

(Q4)

From the graph, we can clearly see that for an increasing number of clauses, the accuracy decreases. Also, the corresponding running time also increases. This can be explained by saying that as the number of clauses increases, the ability to satisfy more and more of them simultaneously becomes increasingly difficult. They may occur cases where for a particular assignment, some clauses satisfy while some clauses fail. And this can happen for two or more clauses simultaneously, making it impossible to satisfy completely. This sentence would be called unsatisfiable, and hence can never give 100% accuracy, eating up both time and resources.

Also, while producing the next generation, we can stop at what we call a local maximum. For example, we reach an assignment that gives us 96% accuracy, the probability that the children also have an almost identically matching assignment increases. Hence the algorithm can have a problem where it thinks that it has reached the maximum fitness possible, but there may be a global maximum somewhere, which may give an accuracy of more than 96%.

(Q5)

There were various difficulties that I faced while optimizing my algorithm. Here are some that are listed.

- **More number of variables:** Again as mentioned above, as the number of variables increase, clashes between them also increase. This unsatisfiable sentence can saturate the algorithm causing it to sometimes give lower than optimal answers because of unnecessary generations to reach a high fitness value.
- **Local Maxima:** In the question above we properly discussed what problems a local maximum can cause. It increases the running time and also prevents the algorithm to reach the most optimal solution possible. Hence we need to take various steps for preventing it. Normally this can be somewhat reduced by implementing various ways such as:
    - If for a certain number of iterations, the accuracy is stagnant at a single value, we can generate some random population again and combined it with a ratio of the current generation, we can speed up this process. This would basically help the algorithm escape the local maxima.

    - Using what we call “early stopping”, where the number of generations that should be generated is a fixed number. This can possibly save a lot of time.
- **Large requirement of time and resources:** Also for our problem, I have used a population size of 20. As this number increases, the time and resources needed for it to properly reach an answer increases by a lot. Yet, it still does not guarantee that we reach an optimal solution.
- **Fine Tuning:** Although there are a lot of strategies that have been set up to tackle these problems, it is a pain-stacking process to try different values and strategies and deciphering the problems that may occur.